NAWCWPNS TP 8341

Unitary (Normal Preserving) Methods for Solving the Schrodinger Equation With Implementation in C and C++

by
Phuc Tran and Fernando J. Escobar
Research and Technology Group

JANUARY 199

NAVAL AIR WARFARE CENTER WEAPONS DIVISION
CHINA LAKE, CA 93555-6100

Naval Air Warfare Center Weapons Division

FOREWORD

This report reviews work completed during fiscal year 1997 on an In-House Laboratory Independent Research project. The project support the research activities of the Computational Sciences Branch, Code 4B4000D and Code 471AF0D, Naval Air Warfare Center, China Lake, California. Funding was provided by ONR's In-House Laboratory Independent Research program; the sponsor is Dr. Ron Derr. The project's goal and accomplishments are provided in the introduction.

This report was reviewed for technical accuracy by Dr. J. Merle Elson.

Approved by
R. L. DERR, *Head*
Research and Technology Group
30 January 1997

Under authority of
J. V. CHENEVEY
RADM, U.S. Navy
Commander

Released for publication by
S. HAALAND
Director for Research and Engineering

NAWCWPNS Technical Publication 8341

Published by . . . . . Technical Information Division
Collation . . . . . Cover, 20 leaves
First printing . . . . . 34 copies

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1997	Final—January 1997

4. TITLE AND SUBTITLE
Unitary (Normal Preserving) Methods for Solving the Schrodinger Equation With Implementation in C and C++

5. FUNDING NUMBERS
N0001497WX20271

6. AUTHOR(S)
Phuc Tran, Fernando J. Escobar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Air Warfare Center Weapons Division
China Lake, CA 93555-6100

8. PERFORMING ORGANIZATION REPORT NUMBER
NAWCWPNS TP 8341

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Ronald L. Derr
Naval Air Warfare Center Weapons Division
Code 4B0000D
China Lake, CA 93555-6100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT
A Statement: distribution unlimited

12B. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

(U) Currently, the Naval Air Warfare Center Weapons Division is exploring using the solution of the time-dependent Schrodinger for some applications in signal processing. The approach being taken to solve this problem is a difference equation. However, the solution based upon a difference equation is unstable, and the solution must be renormalized every time step. This report reviews two methods to obtain solutions that are stable and unitary (preserve the norm). Both methods are based upon the split-operator approach. One method, called the *k-space* method, will use the FFT, and one method, called the *R-space* method, will not. The *k*-space method is more accurate than the *R-space* method. However, it is a "global" method (because it uses the FFT), while the *R*-space method is "local". This difference can be exploited to efficiently use the available computing architecture. Included in this report are two software packages (one in C and one in C++) that implement these methods.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Time-Dependent Schrodinger Equation, Split-Operator Techniques	40
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

CONTENTS

INTRODUCTION

Currently, the Naval Air Warfare Center Weapons Division (NAWCWPNS) is exploring using the solution of the time-dependent Schrodinger for some applications in signal processing. The approach being taken to solve this problem is a difference equation. However, the solution based upon a difference equation is unstable, and the solution must be renormalized every time step. This report reviews two methods to obtain solutions that are stable and unitary (preserve the norm). Both methods are based upon the split-operator approach. One method, called the *k-space* method, will use the fast Fourier transform (FFT), and one method, called the *R-space* method, will not. The *k*-space method is more accurate than the *R-space* method. However, it is a "global" method (because it uses the FFT), while the *R*-space method is "local." This difference can be exploited to efficiently use the available computing architecture. Included in this report are two software packages (one in C and one in C++) that implement these methods. The outline of this report is as follows: section I describes the methods for calculating the time evolution of the wave function governed by a time dependent Schrodinger equation, the Potential Section describes the potentials that are used in the Schrodinger equation, and the Software Packages Section contains the two software listings.

THE TIME-DEPENDENT SCHRODINGER EQUATION

The equation that governs the time evolution of our wave function is the time-dependent Schrodinger equation

$$i\hbar\frac{\partial\psi(x,t)}{\partial t} = H\psi(x,t) = \frac{-\hbar^2\nabla^2}{2m}\psi(x,t) + V(x,t)\psi(x,t) \tag{1}$$

where $V(x,t)$ is the potential. Given the initial wave function we can get the evolution of the wave function in time by integrating Equation 1 over a small time step where the Hamiltonian H is approximately constant. In such case the solution is straightforward

$$\psi(x,t+\Delta t) = e^{-iH\Delta t/\hbar}\psi(x,t). \tag{2}$$

This is repeated to get the time evolution for all time. Note that Equation 2 is unitary if the Hamiltonian is real, i.e. the probability is conserved, and we do not need to renormalize the

wave function after each time step. The right hand side of Equation 2 is evaluated as follows,

$$e^{-iH\Delta t/\hbar}\psi(x,t) = e^{-iV\Delta t/2\hbar}e^{i\hbar^2\nabla^2\Delta t/2\hbar m}e^{-iV\Delta t/2\hbar}\psi(x,t) + O(\Delta t^3). \qquad (3)$$

This can be seen by expanding the exponential on both sides of Equation 3 and compare term by term. Equation 3 is the essence of the split-operator approach.

THE *k-SPACE* METHOD

The first and last exponential in Equation 3 are diagonal in real space and can be evaluated in a straightforward manner. The middle exponential, which contains the Laplacian, can be evaluated accurately and most efficiently in k space. Therefore the steps for evaluating Equation 3 are as follows:

a. Multiply the first exponential with the wave function,
b. Fourier transform the resulting product to k-space using FFT,
c. Multiply with the middle exponential,
d. Fourier transform back to real space,
e. Multiply with the last exponential.

Note that the use of the Fourier transform on a finite grid means that we are using periodic boundary condition.

THE *R-SPACE* METHOD

The disadvantage of the *k-space* method is that the FFT is a global operation and therefore cannot be used to exploit parallel computing architectures (particularly a distributed environment). The *R-space* method (Reference 1) is intended to remedy this deficiency. To evaluate the middle exponential operator we use the same split-operator trick. Let us denote the Laplacian operator as $(L/\Delta x^2)$, which can be written as a sum of 2 parts: A and B. We then have

$$e^{i\hbar L\Delta t/(2m\Delta x^2)} = e^{i\hbar A\Delta t/(4m\Delta x^2)}e^{i\hbar B\Delta t/(2m\Delta x^2)}e^{i\hbar A\Delta t/(4m\Delta x^2)} + O(\Delta t^3). \qquad (4)$$

To understand how and why we do this it is easier to look at a simple example of an 8-point grid. In this case the Laplacian operator, using a three-point formula and periodic boundary condition, has the form

$$L = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}. \tag{5}$$

The Laplacian L can be written as the sum of two matrices A and B as follows

$$L = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix} + \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}. \tag{6}$$

Notice that A and B are 2 x 2 block diagonal; therefore, the exponentiation of these operators can be evaluated easily. The 2 x 2 matrix that we need to diagonalize is

$$M = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{7}$$

This matrix has eigenvalues of 0 and -2. The matrix that diagonalize M is

$$S = S^{-1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{8}$$

That is

$$SMS^{-1} = \begin{pmatrix} 0 & 0 \\ 0 & -2 \end{pmatrix} \equiv \Lambda. \tag{9}$$

The block diagonal nature of the matrices makes Equation 4 very easy to evaluate. The problem of diagonalizing a NxN matrix is reduced to the problem of diagonalizing (N/2) identical 2 x 2 matrices (which we already done in Equations 7 through 9). This algorithm is highly parallel as the 2 x 2 blocks can be done in parallel. For each 2 x 2 block we have

$$e^{-M} = S^{-1}Se^{-M}S^{-1}S = S^{-1}e^{-\Lambda}S \ . \tag{10}$$

A final note about the *R-space* method is that it can be generalized to higher dimension as the Laplacian operator for each dimension commutes among themselves (i.e. $L_x L_y = L_y L_x$).

THE POTENTIAL

THE BASIC POTENTIAL

In the Time-Dependent Schrodinger Equation Section, we discussed the solution to the Schrodinger equation for an arbitrary potential. In this section we briefly describe the types of potential available in the software packages. This section is intended to provide some scientific background to the problem. The programs allow the specification of 1 or 2 potentials. The parameters to be specified for each potential are:

1. Its center
2. Its width
3. Its velocity
4. Its depth (height)
5. Its shape: Square or $1/(\cosh^2)$

Further description of these parameters can be found in the Software Packages Section. Here we want to give some details regarding item 5. The use of a $1/\cosh^2$ potential will help us verify the program as well as gain insight into the physical process (for actual application) since the bound states of this potential are known. This potential has the form $V(x) = -V_0/\cosh^2(x/w)$. We will just state the results for the bound states here; readers interested in the full solution should read Reference 2. The number of bound states for this potential is $n+1$ where n is the largest integer smaller than s with

$$s = \frac{1}{2}\left[-1 + \sqrt{1 + \frac{8mV_o w^2}{\hbar^2}}\right]. \tag{11}$$

The bound states are

$$\psi_n(x) = \left(1-\xi^2\right)^{(s-n)/2} F(-n, 2s-n+1, s-n+1; (1-\xi)/2) \tag{12}$$

where ξ = tanh(x/w), n = 0, 1, 2, ..., and F is the hypergeometric function

$$F(a,b,c;x) = 1 + \frac{ab}{c}x + \frac{a(a+1)b(b+1)}{2\cdot c(c+1)}x^2 + \frac{a(a+1)(a+2)b(b+1)(b+2)}{2\cdot 3\cdot c(c+1)(c+2)}x^3 + \ldots \tag{13}$$

The energy of the nth bound state is

$$E(n) = -\frac{\hbar^2}{2mw^2}(s-n)^2. \tag{14}$$

From Equation 11 we can choose the potential parameters, w and V_o, and the mass m so that the well has only one bound state. Since the mass m is arbitrary we define it in terms of a new variable β,

$$\beta\frac{\hbar^2 k_{\max}^2}{2m} = V_o. \tag{15}$$

The variable β is the ratio of the maximum potential energy to the maximum kinetic energy. For a grid with N points and length L, $k_{max} = (\pi N/L)$.

The program has an option in which one can specify the initial wave function to be a constant or the lowest bound state (as calculated from Equation 12). When the bound state is chosen with appropriate setting of the potential parameters (one potential with a $1/\cosh^2$ shape and zero velocity), the wave function should remain in this state for all time. This can be verified by computing the overlap of the wave function with this bound state as a function of time.

In addition to the real potential, the program allows the addition of an absorptive (imaginary) potential at both ends of the grid. The purpose of this potential is to cause the wave function to go to zero at the boundary of the grid and prevent aliasing or wrap-around effects of the FFT. Unfortunately, this potential also scatters some of the wave function back. An ideal situation is to use some transmitting boundary condition without reflection, but this is not known for the Schrodinger equation at present.

HILDE FILTER

Besides the basic potential one can do the following operations on the potential before using it in the Schrodinger equation:

1. Add random noise.

2. Pass it through a **Hilde filter** (developed by Jeff Hilde, NAWCWPNS), to be rectified, amplified, and convolved with a Gaussian.

Figure 1 shows the flow diagram for the whole program, and Figure 2 shows the steps to generate the potential for use in the Schrodinger equation.

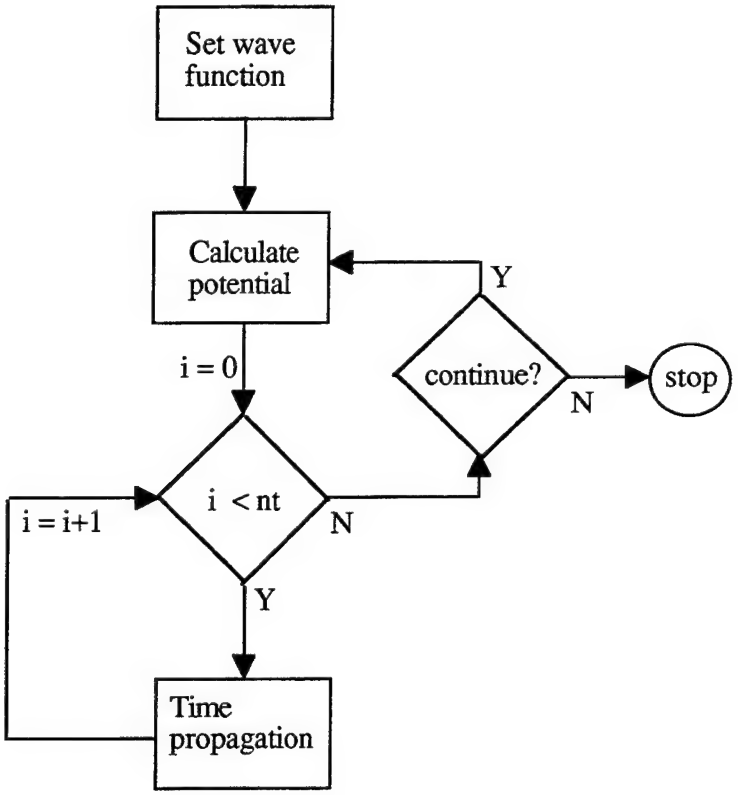

FIGURE 1. Flow Chart of the Program.

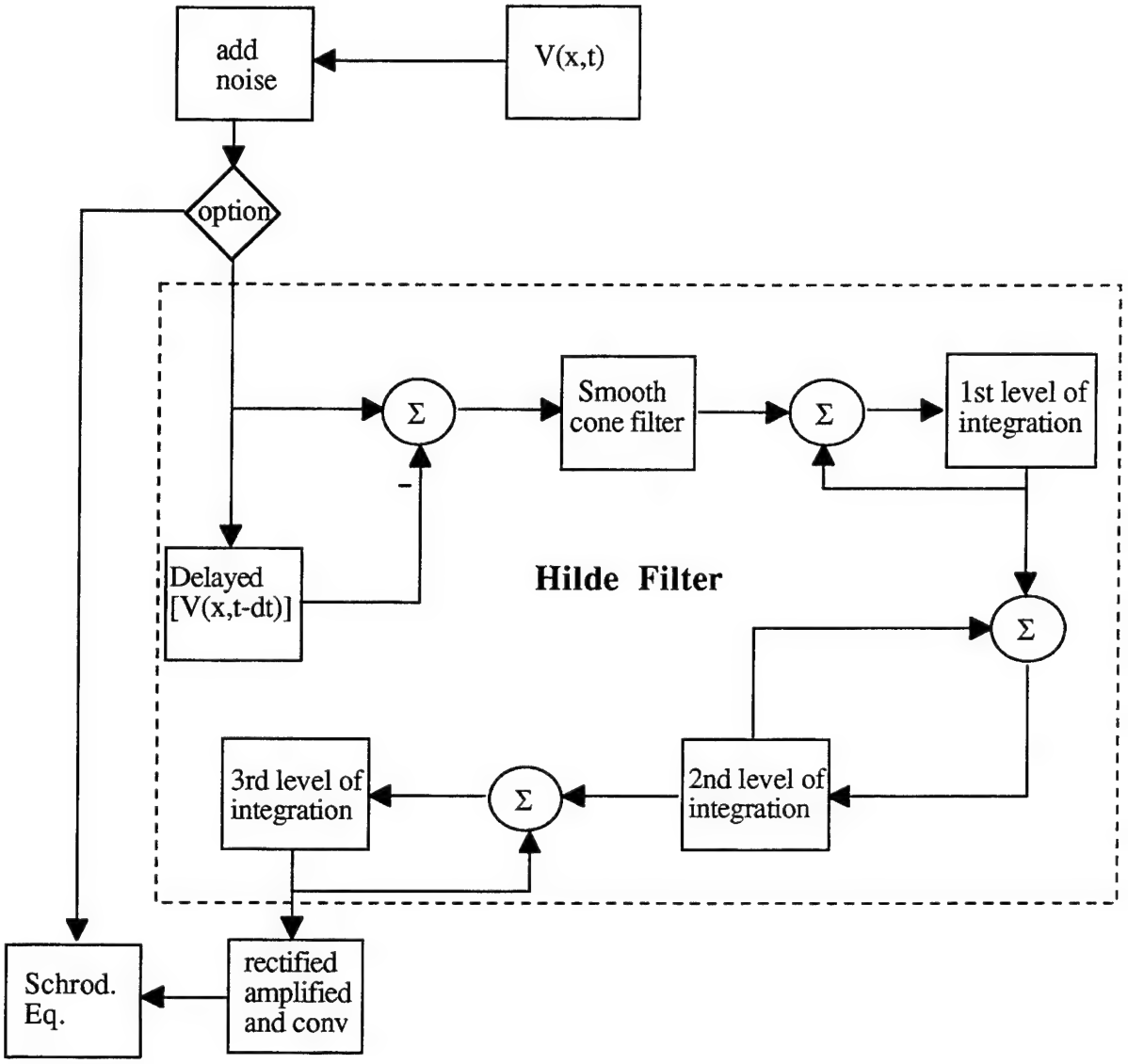

FIGURE 2. Flow Chart for the Generation of the Potential for the Schrodinger Equation.

SOFTWARE PACKAGES

This section contains two source codes: one in C and one in C++. The code in C was developed with Borland C++ (4.5) compiler running on a 486 PC. If you are not developing your own code then this code is adequate. All the changeable parameters (with

comments/descriptions in bold) are at the beginning of the program. This program will give a graphical output showing the time evolution of the potential and the wave function.

If you are developing your own code to solve the Schrodinger equation for a potential of your choice then all you want is the time propagation function. This function, in its most useful form, is found in the second program written in C++. To demonstrate how it works, it was wrapped within a program similar to the one written in C (but without the graphical outputs). This program was developed with a C++ compiler running on an SGI. The time propagation function has 3 parts: the FFT function **four1**, the function **Smultiply**, and the function **time_stepping**. You can substitute your own FFT routine if you desire. These three functions will solve the Schrodinger equation using either the *k-space* or *R-space* method.

Both programs can be obtained from the authors if you do not feel like retyping the program.

C PROGRAM

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <dos.h>
#include <fstream.h>

// for random number generator
#define M1 259200
#define IA1 7141
#define IC1 54773
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349

// global variables
#define nsize 128              // Number of grid points (pixels)
#define Hfactor 0.8            // This is to tune the Hilde filter

float v[nsize], v0[nsize], hil[nsize], hrec[nsize], vnoise[nsize];
float psi1r, psi1i, psi2r, psi2i;
float k2m[nsize] , potr[nsize], poti[nsize], dat[2*nsize];
float ground_state[nsize] ;

float pi = 3.141592653589 ;
```

```
float gain = 80. ;              // For hilde filter
float dt = .05 ;                // Size of time step
int nt = 20 ;                   // Number of time step between frame
long seed = 3411 ;              // Seed for random number generator

/* Whenever the mass is changed (make smaller) remember to
check that the time step dt is adequate !!!!!!!! */

float beta = 0.01 ;             // Mass parameter
float length = 12.8 ;           // Length of grid

// Options
int nopt = 0 ;            // 1 = 1 potential, 0 = 2 potentials
int iopt = 0 ;            // 1 = square, 0 = 1/cosh^2 potential
int potopt = 0 ;          // = 1 use hilde filter, = 0 don't
int wopt = 0 ;            // 1 = const. psi(t=0), 0 = ground state
int output = 0 ;          // Output overlap data to a file: 1 = yes
int kspace = 1 ;          // Using k-space (=1) or R-space (=0)

// Data about potentials
float vel = 0 ;                 // Vel. of potential (pixel/frame)
float xcenter = 55 ;            // Center of potential initially
float vdepth = 0.5 ;            // Depth of potential
float width = 4 ;               // Width of potential in pixels
float vel2 = 0 ;                // Velocity of 2nd potential
float xcenter2 = 70 ;           // Center of 2nd potential initially
float vdepth2 = 1.0 ;           // Depth of 2nd potential
float width2 = 4 ;              // Width of 2nd potential in pixels

float wa = 4 ;                  // Width of absorbing potential
float Va = 10 ;                 // Depth of potential, 0 = no pot.

float noise_factor = 1 ;        // 0 = no noise , 1 = yes
float snr = 5 ;                 // Signal to noise

//x-y coordinate and scale of the plots
int x_target = 30 ;             // Position of potential
int y_target = 10 ;
float scale_target = 5 ;
int x_noise = 30 ;              // Position of (potential +noise)
int y_noise = 100 ;
float scale_noise = 5 ;
int x_hilde = 30;               // Pos. of output from Hilde filter
int y_hilde = 200;
float scale_hilde = 100;
int x_pot = 30 ;                // Pos. of the Schrodinger potential
int y_pot = 250;
float scale_pot = 5 ;
int x_prob = 310;               // Position of probability
int y_prob = 200;
float scale_prob = 100 ;
```

```
//Gaussian for convolution
float gauss[9] = {54,129,242,352,399,352,242,129,54} ;
// Cone Filter
float cone[9] = {1.,2.,3.,4.,5.,4.,3.,2.,1.} ;

// Random number generator (Adapted from "Numerical
Recipes...by Press et. al.")
float ran1(long idum)
{
      static long ix1, ix2, ix3 ;
      static float r[98] ;
      float temp ;
      static int iff=0 ;
      int j ;

      if (idum < 0 || iff == 0)
      {
            iff = 1 ;
            ix1 = (IC1 - idum) % M1 ;
            ix1 = (IA1*ix1 + IC1) % M1 ;
            ix2 = ix1 % M2 ;
            ix1 = (IA1*ix1 + IC1) % M1 ;
            ix3 = ix1 % M3 ;
            for (j=1; j<=97; j++)
            {
                  ix1 = (IA1*ix1 + IC1) % M1 ;
                  ix2 = (IA2*ix2 + IC2) % M2 ;
                  r[j] = (ix1 + ix2*RM2)*RM1 ;
            }
            idum = 1 ;
      }
      ix1 = (IA1*ix1 + IC1) % M1 ;
      ix2 = (IA2*ix2 + IC2) % M2 ;
      ix3 = (IA3*ix3 + IC3) % M3 ;
      j = 1 + ((97*ix3)/M3) ;
      temp = r[j] ;
      r[j] = (ix1+ix2*RM2)*RM1 ;
      return temp ;
}

// Graphics function
void Draw(int x, int y, float d[nsize], float scale)
{
      int i,j;
      j = x;
      moveto(x,y-d[0]*5*scale);
      for (i=1; i<nsize; i++)
      {
            j = j+ (512 / nsize/2);
            lineto(j,y-d[i]*5*scale);
```

```
        }
}

// Convolve input f(x) with a filter g(x) to get h(x)
void Conv(float f[nsize], float g[], float h[nsize], int gsize)
{
        int i, j ;
        float sum ;

        for (i=0; i<nsize; i++)
        {
                sum = 0. ;
                for (j=-(gsize-1); j < gsize; j++)
                {
                        sum = sum + g[j+gsize-1] * f[(nsize+i+j)%nsize] ;
                }
                h[i] = sum ;
        }
}

// Hilde filter
void hilde(float x1[nsize], float x0[nsize], float out[nsize], int flag)
{
        static float diff[nsize], a1[nsize], a2[nsize], a3[nsize] ;
        int i ;

        if (flag == 1)
        {
                for (i=0; i<nsize; i++)
                {
                        a1[i] = 0 ;
                        a2[i] = 0 ;
                        a3[i] = 0 ;
                }
        }

        for (i=0; i< nsize; i++)
        {
                diff[i] = x1[i] - x0[i] ;
        }
        // convolution with a cone filter
        Conv(diff,cone,out,5) ;

        for (i=0; i<nsize; i++)
        {
                a1[i] = Hfactor*a1[i] + (1-Hfactor)*out[i] ;
                a2[i] = Hfactor*a2[i] + (1-Hfactor)*a1[i] ;
                a3[i] = Hfactor*a3[i] + (1-Hfactor)*a2[i] ;
                out[i] = a3[i] ;
                x0[i] = x1[i] ;
        }
```

```
}

// rectify the output from the Hilde filter
void rect(float in[nsize], float out[nsize])
{
      int i ;
      float tem[nsize] ;

      // Rectification
      for (i=0 ; i<nsize; i++)
      {
            if (in[i] < 0)
                  tem[i] = gain*in[i] ;
            else
                  tem[i] = 0 ;
      }

      // convolve with a gaussian 3 times
      Conv(tem,gauss,out,5) ;
      Conv(out,gauss,tem,5) ;
      Conv(tem,gauss,out,5) ;
}

// fft routine (Adapted from "Numerical Recipes...by Press et.
al.")
void four1(float data[], int nn, int isign)
{
      int n, mmax, m, j, istep, i ;
      double wtemp, wr, wpr, wpi, wi, theta ;
      float tempr, tempi, tem ;

      n = nn << 1 ; //n = nn*2
      j = 1 ;
      for (i=1; i<n; i+=2)
      {
            if (j > i)
            {
                  tem = data[i-1] ;
                  data[i-1] = data[j-1] ;
                  data[j-1] = tem ;
                  tem = data[i] ;
                  data[i] = data[j] ;
                  data[j] = tem ;
            }
            m = n >> 1 ; //m = n/2
            while (m>=2 && j > m)
            {
                  j -= m ;    //j = j - m
                  m >>= 1 ; //m = m/2
            }
            j += m ; //j = j+m
```

```
        }
        mmax = 2 ;
        while (n > mmax)
        {
              istep = 2*mmax ;
              theta = 6.28318530717959/(isign*mmax) ;
              wtemp = sin(0.5*theta) ;
              wpr = -2.0*wtemp*wtemp ;
              wpi = sin(theta) ;
              wr = 1.0 ;
              wi = 0.0 ;
              for (m=1; m < mmax; m += 2)
              {
                    for (i=m; i<=n; i += istep)
                    {
                          j = i + mmax ;
                          tempr = wr*data[j-1] - wi*data[j] ;
                          tempi = wr*data[j] + wi*data[j-1] ;
                          data[j-1] = data[i-1] - tempr ;
                          data[j] = data[i] - tempi ;
                          data[i-1] += tempr ;
                          data[i] += tempi ;
                    }
                    wr = (wtemp=wr)*wpr - wi*wpi + wr ;
                    wi = wi*wpr + wtemp*wpi + wi ;
              }
              mmax = istep ;
        }
        //if isign = -1 multiply by 1/nn
        if (isign == -1)
        {
              for (i=0; i < 2*nn; i++)
              {
                    data[i] = data[i]/nn ;
              }
        }
}

//  function to multiply with the 2x2 S-matrix
void Smultiply()
{

        double temp1r, temp1i, temp2r, temp2i ;
        double s11 = 1./sqrt(2.);
        double      s12 = 1./sqrt(2.);
        double      s21 = 1./sqrt(2.);
        double      s22 = -1./sqrt(2.);

        temp1r = s11*psi1r + s12*psi2r ;
        temp1i = s11*psi1i + s12*psi2i ;
        temp2r = s21*psi1r + s22*psi2r ;
```

```
      temp2i = s21*psi1i + s22*psi2i ;
      psi1r = (float)(temp1r) ;
      psi2r = (float)(temp2r) ;
      psi1i = (float)(temp1i) ;
      psi2i = (float)(temp2i) ;
}

// Time propagation function
void time_stepping(float psir[nsize], float psii[nsize], int nn, float
mass)
{
      int i, j ;
      float alpha, temr, temi, dx, tem2r, tem2i ;

      dx = length/nsize ;

      // applied exp[-iVt/2] to the wave function
      for (i=0; i<nn ; i++)
      {
            alpha = potr[i]*dt/2 ;
            temr = cos(alpha)*psir[i] + sin(alpha)*psii[i] ;
            temi = cos(alpha)*psii[i] - sin(alpha)*psir[i] ;
            psir[i] = temr*exp(-poti[i]*dt/2) ;
            psii[i] = temi*exp(-poti[i]*dt/2) ;
      }

      if (kspace)
      {
            // first fft to k space
            for (i=0; i<nn ; i++)
            {
                  j = 2*(i) ;
                  dat[j] = psir[i] ;
                  dat[j+1] = psii[i] ;
            }
            four1(dat,nn,1) ;
            for (i=0; i <nn ; i++)
            {
                  j = 2*(i) ;
                  psir[i] = dat[j] ;
                  psii[i] = dat[j+1] ;
            }

            // applied exp[-i(k**2)t/2m] to the wave function
            for (i=0; i<nn ; i++)
            {
                  alpha = k2m[i]*dt/2 ;
                  temr = cos(alpha)*psir[i] + sin(alpha)*psii[i] ;
                  temi = cos(alpha)*psii[i] - sin(alpha)*psir[i] ;
                  psir[i] = temr ;
                  psii[i] = temi ;
```

```
        }

        // fft back to real space
        for (i=0; i<nn ; i++)
        {
              j = 2*(i) ;
              dat[j] = psir[i] ;
              dat[j+1] = psii[i] ;
        }
        four1(dat,nn,-1) ;
        for (i=0; i <nn ; i++)
        {
              j = 2*(i) ;
              psir[i] = dat[j] ;
              psii[i] = dat[j+1] ;
        }
}
else
{
        // everything is done in real space
        // operate  exp[iAdt/(4*m*dx*dx)]
        for (i=0; i<nsize ; i += 2)
        {
              psi1r = psir[i] ;
              psi1i = psii[i] ;
              psi2r = psir[i+1] ;
              psi2i = psii[i+1] ;
              Smultiply() ;
              alpha = dt/4/mass/dx/dx*(-2) ;
              tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
              tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
              psi2r = tem2r ;
              psi2i = tem2i ;
              Smultiply() ;
              psir[i] = psi1r ;
              psii[i] = psi1i ;
              psir[i+1] = psi2r ;
              psii[i+1] = psi2i ;
        }

        // operate  exp[iBdt/(2*m*dx*dx)]
        psi1r = psir[0] ;
        psi1i = psii[0] ;
        psi2r = psir[nsize-1] ;
        psi2i = psii[nsize-1] ;
        Smultiply() ;
        alpha = dt/2/mass/dx/dx*(-2) ;
        tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
        tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
        psi2r = tem2r ;
        psi2i = tem2i ;
```

```
        Smultiply() ;
        psir[0] = psi1r ;
        psii[0] = psi1i ;
        psir[nsize-1] = psi2r ;
        psii[nsize-1] = psi2i ;

        for (i=1; i<nsize-1 ; i += 2)
        {
              psi1r = psir[i] ;
              psi1i = psii[i] ;
              psi2r = psir[i+1] ;
              psi2i = psii[i+1] ;
              Smultiply() ;
              alpha = dt/2/mass/dx/dx*(-2) ;
              tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
              tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
              psi2r = tem2r ;
              psi2i = tem2i ;
              Smultiply() ;
              psir[i] = psi1r ;
              psii[i] = psi1i ;
              psir[i+1] = psi2r ;
              psii[i+1] = psi2i ;
        }

        // operate  exp[iAdt/(4*m*dx*dx)]
        for (i=0; i<nsize ; i += 2)
        {
              psi1r = psir[i] ;
              psi1i = psii[i] ;
              psi2r = psir[i+1] ;
              psi2i = psii[i+1] ;
              Smultiply() ;
              alpha = dt/4/mass/dx/dx*(-2) ;
              tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
              tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
              psi2r = tem2r ;
              psi2i = tem2i ;
              Smultiply() ;
              psir[i] = psi1r ;
              psii[i] = psi1i ;
              psir[i+1] = psi2r ;
              psii[i+1] = psi2i ;
        }
}

// applied exp[-iVt/2] to the wave function again
for (i=0; i<nn ; i++)
{
      alpha = potr[i]*dt/2 ;
      temr = cos(alpha)*psir[i] + sin(alpha)*psii[i] ;
```

```
            temi = cos(alpha)*psii[i] - sin(alpha)*psir[i] ;
            psir[i] = temr*exp(-poti[i]*dt/2) ;
            psii[i] = temi*exp(-poti[i]*dt/2) ;
      }
}

// function to generate the potential
void potential_generation(long int t)
{
      double xupper, xlower, x, xupper2, xlower2, y, x2, y2 ;
      int i ;

      if (nopt)          // option for 1 or 2 wells
      {
            if (iopt)   // option for square or 1/cosh^2 well
            {
                  // set up square potential
                  xlower = vel*t - width/2 + xcenter ;
                  xupper = vel*t + width/2 + xcenter ;
                  xlower = fmod((xlower+nsize) , nsize) ;
                  xupper = fmod((xupper+nsize) , nsize) ;
                  for (i=0 ; i < nsize ; i++)
                  {
                     v[i] = 0 ;
                     if (xlower < xupper)
                     {
                        if (i >= xlower && i <= xupper) v[i] = -vdepth;
                     }
                     else
                     {
                        if (i <= xupper || i >= xlower) v[i] = -vdepth;
                     }
                  }
            }
            else
            {
                  //  set up 1/cosh**2 potential
                  for (i=0; i<nsize; i++)
                  {
                     x = i - xcenter - vel*t ;
                     y = (cosh(x/width)) ;
                     v[i] = -vdepth/y/y ;
                  }
            }
      }
      else
      {
            if (iopt)   // option for square or 1/cosh^2 well
            {
                  // square well
                  xlower = vel*t - width/2 + xcenter ;
```

```
                xupper = vel*t + width/2 + xcenter ;
                xlower = fmod((xlower+nsize) , nsize) ;
                xupper = fmod((xupper+nsize) , nsize) ;

                xlower2 = vel2*t - width2/2 + xcenter2 ;
                xupper2 = vel2*t + width2/2 + xcenter2 ;
                xlower2 = fmod((xlower2+nsize) , nsize) ;
                xupper2 = fmod((xupper2+nsize) , nsize) ;

                for (i=0 ; i < nsize ; i++)
                {
                   v[i] = 0 ;

                    // check if point falls into 1st well
                   if (xlower < xupper)
                   {
                      if (i >= xlower && i <= xupper) v[i] = -vdepth;
                   }
                   else
                   {
                      if (i <= xupper || i >= xlower) v[i] = -vdepth;
                   }
                    // check to see if point falls into 2nd well
                   if (xlower2 < xupper2)
                   {
                      if (i >= xlower2 && i <= xupper2). v[i] =
                      -vdepth2;
                   }
                   else
                   {
                      if (i <= xupper2 || i >= xlower2) v[i] =
                      -vdepth2;
                   }
                }
          }
          else
          {
                // set up 1/cosh**2 potential
                for (i=0; i<nsize; i++)
                {
                      x = i - xcenter - vel*t ;
                      y = (cosh(x/width)) ;
                      x2 = i - xcenter2 - vel2*t ;
                      y2 = (cosh(x2/width2)) ;
                      v[i] = - vdepth/y/y - vdepth2/y2/y2 ;
                }
          }
     }
}

// add noise to potential
```

```
void add_noise(float snr, long idum)
{
      float x, y ;
      int i ;

      x = pow(10.,snr/10.) ;
      y = vdepth/x ;

      for (i=0; i<nsize; i++)
      {
            vnoise[i] = v[i] + (ran1(idum)-0.5)*2*y*noise_factor ;
      }
}

// Main program
void main(void)
{
      int i, flag, it ;
      long int t;
      float  sum, dk, mass, fact1, fact2, fr, fi, res ;
      float temp[nsize], psir[nsize], psii[nsize], prob[nsize] ;
      double z, ss, yy ;
      long idum ;

      idum = seed ;

      ofstream fout("overlap.txt") ;
      int page=1;
      int idelay=0;

      /* request autodetection */

      int gdriver = EGA, gmode = EGAHI, errorcode;

      /* initialize graphics mode */
      initgraph(&gdriver, &gmode, "c:\\bc45\\bgi");


      /* read result of initialization */
      errorcode = graphresult();

      if (errorcode != grOk)     /* an error occurred */
      {
            printf("Graphics error: %s\n", grapherrormsg(errorcode));
            printf("Press any key to halt:");
            getch();
            exit(1);                /* return with error code */
      }

      // normalize gaussian
      sum = 0 ;
```

```
for (i=0; i<9; i++)
      sum = sum + gauss[i] ;
for (i=0; i<9; i++)
      gauss[i] = gauss[i]/sum ;

// normalize cone filter
sum = 0 ;
for (i=0; i<9; i++)
      sum = sum + cone[i] ;
for (i=0; i<9; i++)
      cone[i] = cone[i]/sum ;

// calculate the mass from beta (a ratio of PE to KE)
mass = beta*(pi*nsize/length)*(pi*nsize/length)/2/vdepth ;

// set up the kinetic energy vector
for (i=1; i<nsize+1 ; i++)
{
      dk = 2*pi/length ;
      k2m[i-1] = dk*(i-1) ;
      if (i > (nsize/2+1))
      {
            k2m[i-1] = dk*(i-nsize-1) ;
      }
      k2m[i-1] = k2m[i-1]*k2m[i-1]/mass ;
}

if (wopt)
{
      // initialize wave function to a constant
      for (i=0 ; i<nsize ; i++)
      {
            psir[i] = 1/sqrt(length) ;
            psii[i] = 0 ;
      }
}
else
{
      // ground state of 1/cosh**2 potential
      for (i=0; i<nsize ; i++)
      {
         z = tanh((i-xcenter)/width) ;
         res = nsize/length ;
         ss = 0.5*(-1+sqrt(1+8*mass*vdepth*width*width/res/res)) ;
         yy = 1 - z*z ;
         psir[i] = pow(yy,ss/2.) ;
         psii[i] = 0 ;
      }
      sum = 0;
      for (i=0; i<nsize ; i++)
      {
```

```
            sum = sum + psir[i]*psir[i] ;
        }
        sum = sqrt(sum) ;
        for (i=0; i<nsize; i++)
        {
              psir[i] = psir[i]/sum  ;
              ground_state[i] = psir[i] ;
        }
}

// set up the absorbing potential at the end of the grid
for (i=0; i<nsize ; i++)
{
        fact1 = i*i/wa/wa ;
        fact2 = (i-nsize+1)*(i-nsize+1)/wa/wa ;
        if (fact1 > 30) fact1 = 30 ;
        if (fact2 > 30) fact2 = 30 ;
        poti[i] = Va * (exp(-fact1) + exp(-fact2)) ;
}

t = 0 ;
for (i=0; i<nsize ; i++)
{
        temp[i] = 0;
}
while (1)      // Beginning of infinite loop
 {
  t++;

  // generate potential
  potential_generation(t) ;

  for (i=0; i<nsize ; i++)
  {
        vnoise[i] = v[i] ;
  }

  // add noise
  add_noise(snr,idum) ;

  // For 1st time step set prev. potential to cur. pot.
  if (t == 1)
  {
        for (i=0; i<nsize; i++)
        {
              v0[i] = vnoise[i] ;
        }
  }

  // call Hilde filter
  flag = 0 ;
```

```
if (t == 1) flag = 1 ;
hilde(vnoise,v0,hil,flag) ;
rect(hil,hrec) ;
for (i=0; i<nsize; i++)
{
      if (potopt)
      {
            potr[i] = hrec[i] ;
      }
      else
      {
            potr[i] = vnoise[i] ;
      }
}

// time stepping for nt steps before updating the pot.
for (it=1 ; it < nt+1 ; it++)
{
      time_stepping(psir,psii,nsize,mass) ;

      // calculate the probability
      for (i=0; i<nsize ; i++)
      {
            prob[i] = psir[i]*psir[i] + psii[i]*psii[i] ;
      }

    // double buffer control
    if (page)
          page = 0;
    else
          page = 1;

    setactivepage(page);
    delay(idelay);
    cleardevice();

    // plot results

    Draw(x_target,y_target,v,scale_target) ;
    Draw(x_target,y_target,temp,scale_target) ;
    Draw(x_noise,y_noise,vnoise,scale_noise) ;
    Draw(x_hilde,y_hilde,hil,scale_hilde) ;
    Draw(x_pot,y_pot,hrec,scale_pot) ;
    Draw(x_prob,y_prob,v,scale_target) ;
    Draw(x_prob,y_prob,temp,scale_prob) ;
    Draw(x_prob,y_prob,prob,scale_prob) ;
    Draw(x_prob,y_prob,potr,scale_pot) ;

    setvisualpage(page);
}
// end "it" loop
```

```
        if (!wopt)
        {
              // calculate overlap with ground state
              sum = 0;
              fr = 0;
              fi = 0;
              for (i=0; i<nsize; i++)
              {
                    sum = sum + psir[i]*psir[i] + psii[i]*psii[i] ;
                    fr = fr + ground_state[i]*psir[i] ;
                    fi = fi + ground_state[i]*psii[i] ;
              }
              if (output)
              {
                    fout << (fr*fr+fi*fi) << "    " << sum << endl ;
              }
        }

        // check for keyboard hit
        if (kbhit())break;

         } // end of infinite loop


        /* clean up */
        getch();
        closegraph();

}
```

C++ PROGRAM

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>

// global variables
const int nsize = 128 ;
float gain = 80. ;            // For hilde filter
float dt = .05 ;              // Size of time step
int nt = 20 ;                 // Number of time step between
updates of                                  // the potential
long seed = 3411 ;            // Seed for random number generator

/* Whenever the mass is changed (make smaller) remember to
check that the time step dt is adequate !!!!!!!! */
float beta = 0.01 ;           // Mass parameter
float length = 12.8 ;         // Length of grid
```

```
//Options
int nopt = 1 ;                  // 1 = 1 potential, 0 = 2 potential
int iopt = 0 ;                  // 1 = square potential,
                                // 0 = 1/cosh^2 potential
int potopt = 0 ;                // If = 1 use hilde filter ,
                                // if = 0 use square (cosh) well
int wopt = 0 ;                  // 1 = constant psi(t=0),
                                // 0 = ground state (for cosh pot)
int output = 1 ;                // Output overlap data: 1 = yes
int kspace = 1 ;                // Use k-space? 1 = yes , 0 = no

//  Data about potentials
float vel = 0 ;                 // Velocity (in pixel per frame)
float xcenter = 55 ;            // Center of potential initially
float vdepth = 1.0 ;            // Depth of potential
float width = 4 ;               // Width of potential in pixels
float vel2 = 0 ;                // Velocity of 2nd potential (in
                                // pixel per frame)
float xcenter2 = 70 ;           // Center of 2nd potential initially
float vdepth2 = 1.0 ;           //  Depth of 2nd potential
float width2 = 4 ;              // Width of 2nd potential in pixels

float wa = 4 ;                  // Width of absorbing potential
float Va = 0 ;                  // Depth of absorbing potential

float noise_factor = 0 ;        // 0 = no noise , 1 = yes
float snr = 5 ;                 // Signal to noise

//  Hilde filter function
/*
   The Hilde filter has the following steps:
        1. Take the difference between this potential and the
           last potential
        2. Convolution with a cone filter (here taken to be 8
           nearest points)
        3. Add a little of this info to prev. info (last loop)
*/


void hilde(float *x1, float *x0, float *out, float *a1, float *a2, float
*a3, int nn)
{
   float cone[9] = {1,2,3,4,5,4,3,2,1} ;
   //normalize cone filter
   float sum ;
   int nf = 5 ; //nf = (size of array "cone" + 1)/2
   sum = 0 ;
   int i;
   for (i=0 ; i<2*nf-1 ; i++)
   {
      sum += cone[i] ;
```

```
    }
    for (i=0 ; i<2*nf-1 ; i++)
    {
       cone[i] /= sum ;
    }

    // Take the difference
    float *diff ;
    diff = new float[nn] ;
    for (i=0 ; i<nn ; i++)
    {
       diff[i] = x1[i] - x0[i] ;
    }

    // Convolution with a cone filter
    int j;
    for (i=0 ; i<nn ; i++)
    {
       sum = 0;
       for (j=-(nf-1); j < nf ; j++)
       {
          sum += cone[j+nf-1]*diff[(nn+i+j)%nn] ;
       }
       out[i] = sum ;
    }

    // Step no. 3. Hfactor is fraction of past info to keep
    float Hfactor = 0.8 ;
    for (i=0 ; i<nn ; i++)
    {
       a1[i] = Hfactor*a1[i] + (1-Hfactor)*out[i] ;
       a2[i] = Hfactor*a2[i] + (1-Hfactor)*a1[i] ;
       a3[i] = Hfactor*a3[i] + (1-Hfactor)*a2[i] ;
       out[i] = a3[i] ;
       x0[i] = x1[i] ;
    }
    delete[] diff;
}

/*
    Function to rectified the output from a Hilde filter.
     also convolve it with a gaussian 3 times.
*/

void rect(float *in, float *out, int nn, float gain)
{
    float gauss[9] = {54,129,242,352,399,352,242,129,54} ;
    //normalize gaussian filter
    float sum ;
    int nf = 5 ;          // nf = (size of array "gauss" + 1)/2
    sum = 0 ;
```

```
    int i;
    for (i=0 ; i<2*nf-1 ; i++)
    {
       sum += gauss[i] ;
    }
    for (i=0 ; i<2*nf-1 ; i++)
    {
       gauss[i] /= sum ;
    }

    float *tem ;
    tem = new float[nn] ;
    float *tem2 ;
    tem2 = new float[nn] ;

    // Rectification and amplification
    for (i=0 ; i<nn ; i++)
    {
       if (in[i] < 0)
          tem[i] = gain*in[i] ;
       else
         tem[i] =  0 ;
    }

    // Convolution with a gaussian filter 3 times
    int j,k;
    for (k=0 ; k<3 ; k++)
    {
       for (i=0 ; i<nn ; i++)
       {
          sum = 0;
          for (j=-(nf-1); j < nf ; j++)
          {
             sum += gauss[j+nf-1]*tem[(nn+i+j)%nn] ;
          }
          tem2[i] = sum ;
       }
       for (i=0 ; i<nn ; i++)
       {
          tem[i] = tem2[i];
       }
    }
    for (i=0 ; i<nn ; i++)
    {
       out[i] = tem[i] ;
    }
    delete[] tem;
    delete[] tem2;
}
```

```
// Random number generator (Adapted from Numerical Recipes...
by Press et. al.)
float ran1(long &idum)
{
   const long  M1 = 259200 ;
   const long IA1 = 7141 ;
   const long IC1 = 54773 ;
   const float RM1 = (1.0/M1) ;
   const long M2 = 134456 ;
   const long IA2 = 8121 ;
   const long IC2 = 28411 ;
   const float RM2 = (1.0/M2) ;
   const long M3 = 243000 ;
   const long IA3 = 4561 ;
   const long IC3 = 51349 ;

   static long ix1, ix2, ix3 ;
   static float r[98] ;
   float temp ;
   static int iff=0 ;
   int j ;

   if (idum < 0 || iff == 0)
   {
      iff = 1 ;
      ix1 = (IC1 - (idum)) % M1 ;
      ix1 = (IA1*ix1 + IC1) % M1 ;
      ix2 = ix1 % M2 ;
      ix1 = (IA1*ix1 + IC1) % M1 ;
      ix3 = ix1 % M3 ;
      for (j=1 ; j<=97; j++)
      {
         ix1 = (IA1*ix1 + IC1) % M1 ;
         ix2 = (IA2*ix2 + IC2) % M2 ;
         r[j] = (ix1 + ix2*RM2)*RM1 ;
      }
      idum = 1 ;
   }
   ix1 = (IA1*ix1 + IC1) % M1 ;
   ix2 = (IA2*ix2 + IC2) % M2 ;
   ix3 = (IA3*ix3 + IC3) % M3 ;
   j = 1 + ((97*ix3)/M3) ;
   temp = r[j] ;
   r[j] = (ix1+ix2*RM2)*RM1 ;
   return temp;
}

// fft routine (Adapted from Numerical Recipes... by Press et.
al.)
/* A section was added at the end to divide the result by 1/N
if the inverse FFT is taken */
```

```
void four1(float *data, int nn, int isign)
{
   int n, mmax, m, j, istep, i ;
   double wtemp, wr, wpr, wpi, wi, theta ;
   float tempr, tempi, tem ;

   n = nn << 1 ; //n = nn*2
   j = 1 ;
   for (i=1; i<n; i+=2)
   {
      if (j > i)
      {
         tem = data[i-1] ;
         data[i-1] = data[j-1] ;
         data[j-1] = tem ;
         tem = data[i] ;
         data[i] = data[j] ;
         data[j] = tem ;
      }
      m = n >> 1 ; //m = n/2
      while (m>=2 && j > m)
      {
         j -= m ;    //j = j - m
         m >>= 1 ; //m = m/2
      }
      j += m ; //j = j+m
   }
   mmax = 2 ;
   while (n > mmax)
   {
      istep = 2*mmax ;
      theta = 6.28318530717959/(isign*mmax) ;
      wtemp = sin(0.5*theta) ;
      wpr = -2.0*wtemp*wtemp ;
      wpi = sin(theta) ;
      wr = 1.0 ;
      wi = 0.0 ;
      for (m=1; m < mmax; m += 2)
      {
         for (i=m; i<=n; i += istep)
         {
            j = i + mmax ;
            tempr = wr*data[j-1] - wi*data[j] ;
            tempi = wr*data[j] + wi*data[j-1] ;
            data[j-1] = data[i-1] - tempr ;
            data[j] = data[i] - tempi ;
            data[i-1] += tempr ;
            data[i] += tempi ;
         }
         wtemp = wr ;
         wr = wr*wpr - wi*wpi + wr ;
```

```
            wi = wi*wpr + wtemp*wpi + wi ;
         }
         mmax = istep ;
      }
      //if isign = -1 multiply by 1/nn
      if (isign == -1)
      {
         for (i=0; i < 2*nn; i++)
         {
            data[i] = data[i]/nn ;
         }
      }
}

// Function to multiply a 2 component vector with a 2x2 matrix
S
void Smultiply(float &psi1r, float &psi1i, float &psi2r, float &psi2i)
{
   double s11 = 1./sqrt(2.) ;
   double s12 = 1./sqrt(2.) ;
   double s21 = 1./sqrt(2.) ;
   double s22 = -1./sqrt(2.) ;
   double tem1r, tem1i, tem2r, tem2i ;

   tem1r = s11*psi1r + s12*psi2r ;
   tem1i = s11*psi1i + s12*psi2i ;
   tem2r = s21*psi1r + s22*psi2r ;
   tem2i = s21*psi1i + s22*psi2i ;

   psi1r = (float)(tem1r) ;
   psi1i = (float)(tem1i) ;
   psi2r = (float)(tem2r) ;
   psi2i = (float)(tem2i) ;
}

//  Time  propagation  function

/*
   parameters  to  be  passed  to  this  functions  are:
        number of grid points = nn
        real  part  of  the  wave  function psi  =  psir[nn]
        im    part  of  the  wave  function psi  =  psii[nn]
       mass = mass
       time step = dt
        real  part  of  potential  =  potr[nn]
        im    part  of  potential  =  poti[nn]
        length of grid = length
        option  to  calculate  in  k  space  or  real  space  =  kspace
*/
```

```
void time_stepping(int nn, float *psir, float *psii, float mass, float
dt, float *potr, float *poti, float length, int kspace)
{
   int i, j ;
   float alpha, temr, temi, dx, tem2r, tem2i ;
   float psi1r, psi1i, psi2r, psi2i ;
   float *dat ;
   dat = new float[2*nn] ;

   // Set up the kinetic energy vector

   float *k2m ;
   k2m = new float[nn] ;
   double pi = 3.141592653589 ;

   float dk ;
   dk = (float)(2*pi/length) ;
   for (i=0; i<nsize ; i++)
   {
      k2m[i] = dk*(i) ;
      if (i > (nsize/2))
      {
         k2m[i] = dk*(i-nsize) ;
      }
      k2m[i] = k2m[i]*k2m[i]/mass ;
   }

   dx = length/nn ;

   // Applied exp[-iVt/2] to the wave function
   for (i=0; i<nn ; i++)
   {
      alpha = potr[i]*dt/2 ;
      temr = cos(alpha)*psir[i] + sin(alpha)*psii[i] ;
      temi = cos(alpha)*psii[i] - sin(alpha)*psir[i] ;
      psir[i] = temr*exp(-poti[i]*dt/2) ;
      psii[i] = temi*exp(-poti[i]*dt/2) ;
   }

   if (kspace)    // Option to use the k-spcae or R-space method
   {
       // First fft to k space
      for (i=0; i<nn ; i++)
      {
         j = 2*(i) ;
         dat[j] = psir[i] ;
         dat[j+1] = psii[i] ;
      }
      four1(dat,nn,1) ;
      for (i=0; i <nn ; i++)
      {
```

```
            j = 2*(i) ;
            psir[i] = dat[j] ;
            psii[i] = dat[j+1] ;
         }

          // Applied exp[-i(k**2)t/2m] to the wave function
         for (i=0; i<nn ; i++)
         {
            alpha = k2m[i]*dt/2 ;
            temr = cos(alpha)*psir[i] + sin(alpha)*psii[i] ;
            temi = cos(alpha)*psii[i] - sin(alpha)*psir[i] ;
            psir[i] = temr ;
            psii[i] = temi ;
         }

          // fft back to real space
         for (i=0; i<nn ; i++)
         {
            j = 2*(i) ;
            dat[j] = psir[i] ;
            dat[j+1] = psii[i] ;
         }
         four1(dat,nn,-1) ;
         for (i=0; i <nn ; i++)
         {
            j = 2*(i) ;
            psir[i] = dat[j] ;
            psii[i] = dat[j+1] ;
         }
      }
      else
      {
          // Everything is done in real space

          // Operate exp[iAdt/(4*m*dx*dx)]
         for (i=0; i<nn ; i += 2)
         {
            psi1r = psir[i] ;
            psi1i = psii[i] ;
            psi2r = psir[i+1] ;
            psi2i = psii[i+1] ;
            Smultiply(psi1r,psi1i,psi2r,psi2i) ;
            alpha = dt/4/mass/dx/dx*(-2) ;
            tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
            tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
            psi2r = tem2r ;
            psi2i = tem2i ;
            Smultiply(psi1r,psi1i,psi2r,psi2i) ;
            psir[i] = psi1r ;
            psii[i] = psi1i ;
            psir[i+1] = psi2r ;
```

```
    psii[i+1] = psi2i ;
}

 // Operate  exp[iBdt/(2*m*dx*dx)]
psi1r = psir[0] ;
psi1i = psii[0] ;
psi2r = psir[nn-1] ;
psi2i = psii[nn-1] ;
Smultiply(psi1r,psi1i,psi2r,psi2i) ;
alpha = dt/2/mass/dx/dx*(-2) ;
tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
psi2r = tem2r ;
psi2i = tem2i ;
Smultiply(psi1r,psi1i,psi2r,psi2i) ;
psir[0] = psi1r ;
psii[0] = psi1i ;
psir[nn-1] = psi2r ;
psii[nn-1] = psi2i ;

for (i=1; i<nn-1 ; i += 2)
{
   psi1r = psir[i] ;
   psi1i = psii[i] ;
   psi2r = psir[i+1] ;
   psi2i = psii[i+1] ;
   Smultiply(psi1r,psi1i,psi2r,psi2i) ;
   alpha = dt/2/mass/dx/dx*(-2) ;
   tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
   tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
   psi2r = tem2r ;
   psi2i = tem2i ;
   Smultiply(psi1r,psi1i,psi2r,psi2i) ;
   psir[i] = psi1r ;
   psii[i] = psi1i ;
   psir[i+1] = psi2r ;
   psii[i+1] = psi2i ;
}

 // Operate  exp[iAdt/(4*m*dx*dx)]
for (i=0; i<nn ; i += 2)
{
   psi1r = psir[i] ;
   psi1i = psii[i] ;
   psi2r = psir[i+1] ;
   psi2i = psii[i+1] ;
   Smultiply(psi1r,psi1i,psi2r,psi2i) ;
   alpha = dt/4/mass/dx/dx*(-2) ;
   tem2r = cos(alpha)*psi2r - sin(alpha)*psi2i ;
   tem2i = cos(alpha)*psi2i + sin(alpha)*psi2r ;
   psi2r = tem2r ;
```

```
            psi2i = tem2i ;
            Smultiply(psi1r,psi1i,psi2r,psi2i) ;
            psir[i] = psi1r ;
            psii[i] = psi1i ;
            psir[i+1] = psi2r ;
            psii[i+1] = psi2i ;
         }
      }

      // Applied exp[-iVt/2] to the wave function again
      for (i=0; i<nn ; i++)
      {
         alpha = potr[i]*dt/2 ;
         temr = cos(alpha)*psir[i] + sin(alpha)*psii[i] ;
         temi = cos(alpha)*psii[i] - sin(alpha)*psir[i] ;
         psir[i] = temr*exp(-poti[i]*dt/2) ;
         psii[i] = temi*exp(-poti[i]*dt/2) ;
      }

      delete[] dat ;
      delete[] k2m ;
   }

   // Function to generate the potential
   void potential_generation(long t, float *v, int nn)
   {
      double xupper, xlower, x, xupper2, xlower2, y, x2, y2 ;
      int i ;
      if (nopt)            // Option for 1 or 2 wells
      {
         if (iopt)         // Option for square or cosh well
         {
             // Set up square potential
            xlower = vel*t - width/2 + xcenter ;
            xupper = vel*t + width/2 + xcenter ;
            xlower = fmod((xlower+nn) , nn) ;
            xupper = fmod((xupper+nn) , nn) ;
            for (i=0 ; i < nn ; i++)
            {
               v[i] = 0 ;
               if (xlower < xupper)
               {
                  if (i >= xlower && i <= xupper) v[i] = -vdepth;
               }
               else
               {
                  if (i <= xupper || i >= xlower) v[i] = -vdepth;
               }
            }
         }
         else
```

```
    {
        // Set up 1/cosh**2 potential
       for (i=0; i<nn; i++)
       {
          x = i - xcenter - vel*t ;
          y = (cosh(x/width)) ;
          v[i] = -vdepth/y/y ;
       }
    }
}
else
{
   if (iopt)          // Option for square or cosh well
   {
       // Square well
      xlower = vel*t - width/2 + xcenter ;
      xupper = vel*t + width/2 + xcenter ;
      xlower = fmod((xlower+nn) , nn) ;
      xupper = fmod((xupper+nn) , nn) ;
      xlower2 = vel2*t - width2/2 + xcenter2 ;
      xupper2 = vel2*t + width2/2 + xcenter2 ;
      xlower2 = fmod((xlower2+nn) , nn) ;
      xupper2 = fmod((xupper2+nn) , nn) ;
      for (i=0 ; i < nn ; i++)
      {
         v[i] = 0 ;
          // Check to see if point falls into 1st well
         if (xlower < xupper)
         {
            if (i >= xlower && i <= xupper) v[i] = -vdepth;
         }
         else
         {
            if (i <= xupper || i >= xlower) v[i] = -vdepth;
         }
          // Check to see if point falls into 2nd well
         if (xlower2 < xupper2)
         {
            if (i >= xlower2 && i <= xupper2) v[i] = -vdepth2;
         }
         else
         {
            if (i <= xupper2 || i >= xlower2) v[i] = -vdepth2;
         }
      }
   }
   else
   {
       // Set up 1/cosh**2 potential
      for (i=0; i<nn; i++)
      {
```

```
            x = i - xcenter - vel*t ;
            y = (cosh(x/width)) ;
            x2 = i - xcenter2 - vel2*t ;
            y2 = (cosh(x2/width2)) ;
            v[i] = - vdepth/y/y - vdepth2/y2/y2 ;
         }
      }
   }
}

// Main program
void main(void)
{
   int i, it ;
   long t;
   float  sum, mass, fact1, fact2, fr, fi, res, x, y ;
   float temp[nsize], psir[nsize], psii[nsize], prob[nsize] ;
   float a1[nsize] , a2[nsize], a3[nsize], ground_state[nsize] ;
   float potr[nsize], poti[nsize], v[nsize], v0[nsize], vnoise[nsize] ;
   float hil[nsize], hrec[nsize] ;
   double z, ss, yy ;
   long idum ;
   double pi = 3.141592653589 ;

   idum = seed ;

   ofstream fout("overlap.txt") ;
   ofstream vout("potential.txt") ;

   // Calculate the mass from beta (a ratio of PE to KE)
   mass = beta*(pi*nsize/length)*(pi*nsize/length)/2/vdepth ;

   if (wopt)
   {
       // Initialize wave function to a constant
      for (i=0 ; i<nsize ; i++)
      {
         psir[i] = 1/sqrt(length) ;
         psii[i] = 0 ;
      }
   }
   else
   {
       // Ground state of 1/cosh**2 potential
      for (i=0; i<nsize ; i++)
      {
         z = tanh((i-xcenter)/width) ;
         res = nsize/length ;
         ss = 0.5*(-1+sqrt(1+8*mass*vdepth*width*width/res/res)) ;
         yy = 1 - z*z ;
         psir[i] = (float)(pow(yy,ss/2.)) ;
```

```
      psii[i] = 0 ;
   }
    // Normalize wave function
   sum = 0;
   for (i=0; i<nsize ; i++)
   {
      sum = sum + psir[i]*psir[i] ;
   }
   sum = sqrt(sum) ;
   for (i=0; i<nsize; i++)
   {
      psir[i] = psir[i]/sum  ;
      ground_state[i] = psir[i] ;
   }
}

 // Set up the absorbing potential at the end of the grid
for (i=0; i<nsize ; i++)
{
   fact1 = i*i/wa/wa ;
   fact2 = (i-nsize+1)*(i-nsize+1)/wa/wa ;
   if (fact1 > 30) fact1 = 30 ;
   if (fact2 > 30) fact2 = 30 ;
   poti[i] = Va * (exp(-fact1) + exp(-fact2)) ;
}

t = 0 ;
for (i=0; i<nsize ; i++)
{
   temp[i] = 0;
   a1[i] = 0 ;
   a2[i] = 0 ;
   a3[i] = 0 ;
   v[i] = 0;
}

for (t=1 ; t< 100 ; t++)
{
   //t++;

    // Generate potential
   potential_generation(t,v,nsize) ;

   for (i=0; i<nsize ; i++)
   {
      vnoise[i] = v[i] ;
   }

    // Add noise
   x = (float)(pow(10.,snr/10.)) ;
   y = vdepth/x ;
```

```
for (i=0 ; i<nsize ; i++)
{
   vnoise[i] = v[i] + (ran1(idum)-0.5)*y*noise_factor ;
}

 // For 1st time step set previous potential to present
// potential
if (t == 1)
{
   for (i=0; i<nsize; i++)
   {
      v0[i] = vnoise[i] ;
   }
}
 // Call Hilde filter
hilde(vnoise,v0,hil,a1,a2,a3,nsize) ;
 // Rectify, amplify, and convolve output of Hilde filter
rect(hil,hrec,nsize,gain) ;

 if (potopt)   // Option to choose a po. for the Schrod. eq.
{
   for (i=0; i<nsize; i++)
   {
      potr[i] = hrec[i] ;
   }
}
else
{
   for (i=0; i<nsize; i++)
   {
      potr[i] = vnoise[i] ;
   }
}

 // Time stepping for nt steps before updating the pot.
for (it=1 ; it < nt+1 ; it++)
{
   time_stepping(nsize,psir,psii,mass,dt,potr,poti,length,kspace);

    // Calculate the probability
   for (i=0; i<nsize ; i++)
   {
      prob[i] = psir[i]*psir[i] + psii[i]*psii[i] ;
   }
}
 // End "it" loop

if (!wopt)
{
    // Calculate overlap with ground state
   sum = 0;
```

```
            fr = 0;
            fi = 0;
            for (i=0; i<nsize; i++)
            {
               sum = sum + psir[i]*psir[i] + psii[i]*psii[i] ;
               fr = fr + ground_state[i]*psir[i] ;
               fi = fi + ground_state[i]*psii[i] ;
            }
            if (output)
            {
               fout << (fr*fr+fi*fi) << "    " << sum << endl ;
            }
         }
      } // End of t loop

      // Output the real part of the potential
      for (i = 0 ; i<nsize ; i++)
      {
         vout << i << "    " << potr[i] << endl ;
      }
}
```

REFERENCES

1. H. De Raedt. *Comput. Phys. Rep.*, Vol. 7, No. 1 (1987).

2. Landau and Lifshitz. *Quantum Mechanics*, 2nd ed. New York, Pergamon Press, 1965. Pp. 72-73.

INITIAL DISTRIBUTION

1 Commander in Chief, U. S. Pacific Fleet, Pearl Harbor (Code 325)
1 Naval War College, Newport
1 Headquarters, 497 IG/INT, Falls Church (OUWG Chairman)
2 Defense Technical Information Center, Fort Belvoir
1 Center for Naval Analyses, Alexandria, VA (Technical Library)

ON SITE DISTRIBUTION

19 Code 4B4000D
- S. Chesnut (1)
- F. Escobar (6)
- D. Gillespie (1)
- P. Tran (10)
- A. Van Nevel (1)

4 Code 4BL000D (3 plus Archives Copy)
2 Code 471AF0D, D. Andes
1 Code 472000D
1 Code 473000D
1 Code 474000D